

The Kepler DB, a database management system for arrays, sparse arrays, and binary data

Sean McCauliff^{*a}, Miles T. Cote^b, Forrest R. Girouard^a, Christopher Middour^a, Todd C. Klaus^a, Bill Wohler^a

^aOrbital Sciences/NASA Ames Research Center, Mail Stop 244-30, Moffett Field, CA 94035

^bNASA Ames Research Center, Mail Stop 244-30 Moffett Field, CA 94035

ABSTRACT

The *Kepler* Science Operations Center stores pixel values on approximately six million pixels collected every 30 minutes, as well as data products that are generated as a result of running the *Kepler* science processing pipeline. The *Kepler Database management system (Kepler DB)* was created to act as the repository of this information. After one year of flight usage, *Kepler DB* is managing 3 TiB of data and is expected to grow to over 10 TiB over the course of the mission. *Kepler DB* is a non-relational, transactional database where data are represented as one-dimensional arrays, sparse arrays or binary large objects. We will discuss *Kepler DB*'s APIs, implementation, usage and deployment at the *Kepler* Science Operations Center.

Keywords: Kepler Mission, database, non-relational, time series, Java, photometry, transactional

1. INTRODUCTION

The *Kepler Mission*^{1,8} uses a space-based photometer with a 115 square degree field of view to search for transit signatures of exoplanets, with a particular emphasis on Earth-size exoplanets in the habitable zones of their parent stars. In order to detect the transits of Earth-sized exoplanets, *Kepler* has recorded the flux from approximately 192,000 distinct stars in its field of view at 30-minute intervals. The focal plane has 42 CCD sensors with two outputs each. These are organized into modules with two sensor chips each. A subset (5.66 M pixels) of the total pixels, those around interesting target stars, are downloaded each month and stored in the *Kepler DB*.

Kepler DB was designed to satisfy uses cases required to operate the *Kepler* Science Operations Center (SOC) science processing pipeline.² This includes ingesting spacecraft data, storing intermediate processing products, interactively retrieving data products for researchers, and storing portable document format (PDF) report files which are served over a web interface.

Prototype implementations of the *Kepler* science processing pipeline used a popular relational database to store array and time series data. Even with significant tuning, a relational database management system would take too long to process light curves generated by a single stage of our pipeline. Initial implementations of the *Kepler DB* showed a significant speedup using a custom database implementation. Unlike a conventional relational database, it supports a limited number of data types that are more suitable for the efficient storage and retrieval of data needed by the *Kepler* science processing pipeline. This is part of a trend of non-relational and column-oriented databases. For example, *SciDB*³ is being developed to store the data of the *Large Synoptic Survey Telescope*. It will support a mostly non-transactional mode of operation over 10s of PiB of storage on nested array structures. The *Kepler DB* has different requirements; storing a more modest amount of data, it supports transactions over an arbitrary number of its Create, Read, Update, Delete (CRUD) operations.

Kepler DB is organized in the traditional client-server model. The client and server communicate using a custom protocol. *Kepler DB* is implemented using the Java programming language, and runs on Fedora

^{*}sean.d.mccauliff@nasa.gov

GNU/Linux in the *Kepler*SOC at NASA Ames Research Center. MATLAB’s Java integration also allows the clients to use the MATLAB programming language to interface with the server. A relational database is used in conjunction with *Kepler DB* to store application objects and heavily reduced science data. *Kepler DB* coordinates transactions using the XA⁹ protocol with external data sources (e.g., JDBC and Java Messaging Service).

Diagrams used in this document may omit unnecessary details like packages, accessor methods, parameter names, parameter types, classes, or method calls when these do not aid the understanding of the reader. In particular, classes that implement abstract methods defined by abstract super classes or interfaces do not list the implemented methods. The UML notation for parameterized classes is not used. The non-normative UML keyword {frozen} has been used in class diagrams to indicate data members that are completely unchanging. In the case of primitive types, these refer to members that use the final keyword. Frozen is only indicated on member objects that are both immutable and have final references. Names of some classes have been changed from the actual implementation classes in the source code.

2. USES

Kepler DB can store several array types: IntArray, FloatArray, and DoubleArray. *Kepler DB* can also store one sparse array type FloatSparseArray and streamed byte arrays (Blobs). These are discussed more in section 3.1.

Kepler DB is primarily used by multiple batch processes running in parallel in one of SOC’s compute clusters. Figure 1 provides a summary of the flow of information in the *Kepler* SOC. The Data Receipt² software component loads *Kepler DB* with raw data from the spacecraft. Example pipeline module components run are Calibration, Photometric Analysis, Pre-search Data Conditioning, Transiting Planet Search and Data Validation. Finally, data is exported from the system using FITS archivers and the Mission Reports Web Application. Science User Tools are interactive MATLAB programs used by scientists to inspect data stored in the *Kepler DB*. Science processing pipeline modules are run in parallel on a cluster. There is one Calibration process for every CCD channel (84), while Data Validation runs with a hundred or more processes. Pipeline modules can be reading or writing data concurrently along with Data Receipt, FITS Archiver, and Science User Tools.

Pixel data from the spacecraft are stored in IntArrays per pixel coordinate. Every pixel coordinate (CCD module, CCD output, row, column) that has been collected has its own IntArray. Pixel data from the spacecraft are collected monthly; only a subset of the pixel coordinates are collected. An individual exposure on the spacecraft is collected about every 6 seconds. These values are co-added over a 30-minute or 1-minute period on board the spacecraft. In a one-month period, 30-minute co-adds grow each Array by about 1.4 Ki values; 1-minute co-adds grow about 42 Ki values per month. Since all pixels are regularly sampled throughout the mission, we are able to map every Array index to a unique time stamp. Every three months the spacecraft rotates in order to keep the solar panels oriented to the sun; a new set of pixel address definitions are uploaded, which means different sets of pixels are collected. This leaves large gaps (contiguous regions of undefined, null data) in the IntArrays; this is represented efficiently (see section 4.1). Ancillary engineering data, readouts of the sensors inside the spacecraft which assess the environmental conditions of the spacecraft, are stored as FloatSparseArrays. Each kind of ancillary engineering data is collected at a different rate, making it impossible to store it in an array classes.

Further processing steps transform the raw pixel values, represented in the analog-to-digital converter’s units, into calibrated photo-electron values and their uncertainties; these are stored in FloatArray objects. The Photometric Analysis component computes a single flux FloatArray from the individual pixel FloatArrays (on average, 32 per star), as well as the centroids of the stars that are stored as DoubleArray objects. Photometric Analysis also has a cosmic ray and outlier detection algorithm that stores these infrequently detected events as FloatSparseArrays. Other pipeline modules generally store their data as FloatArrays (computations in pipeline modules are carried out in double precision, but stored in single precision). One year after launch, there are approximately 87M Arrays and 27M FloatSparseArrays stored in the *Kepler DB* server.

Blobs are typically used to store files in MATLAB’s “.mat” file format or PDF files. The “.mat” files store arbitrary MATLAB data structures, most often large matrices of polynomial coefficients. PDF reports are generated by most pipeline modules. Reports are used by the operators and scientists to assess the health of the

spacecraft as well as inform about potentially interesting scientific phenomena. Some reports are served over the web from the *Kepler DB* via the Mission Reports Web Application.

The Science User Tools are read-only tools; they do not write anything into the *Kepler DB*. Using MATLAB's integration with the Java language, *Kepler DB*'s API methods are called directly to retrieve data. One Science User Tool allows the retrieval of all the data currently stored about a particular target star (raw pixels, calibrated pixels, flux, etc.). Scientists utilizing data stored in the *Kepler DB* run these tools at unscheduled times; as they feel they are needed.

A relational database is often used in conjunction with *Kepler DB*. In our science processing pipeline, there is a significant amount of structured application data associated with tracking how stars are related to pixels: models needed to map from pixels to (right ascension, declination) coordinates, which pixel coordinates were selected for a star, global timestamps, stellar data, etc. A relational database is also used to store application objects like user accounts, roles, and persistent database logging.

3. INTERFACES

3.1 API and Data Model

A user data object is referenced by an *ObjectId*, which is similar to a file system file name. There is a path part and a name part. An example *ObjectId* we use would be `"/cal/pixels/lct/10/1/552:42"`. The semantics of this name are opaque to the *Kepler DB*. The user of this *ObjectId* created it to mean the calibrated pixel ("cal"), which is part of the 30-minute sampled target pixels ("lct") on CCD module 10 CCD output 1 pixel with row 552 and column 42.

The *Kepler DB* provides three data types of user data objects: an *Array*, a *SparseArray* and a *Blob*. *Array* has three subclasses: *IntArray* can store arrays of 32-bit signed integers, *FloatArray* can store 32-bit floating point numbers, and *DoubleArray* can store 64-bit floating point numbers. The *SparseArray* object can store sparse arrays of data with a 64-bit floating point number as the key and a 32-bit floating point number as the value. *Blobs* are just plain files without any kind of indexing.

Each data point or *Blob* stored in *Kepler DB* has an originator identifier (ID) associated with it. The originator ID is a 64-bit signed integer which allows a user to identify the instance of the *Kepler* science processing pipeline task⁷ that created the data.

Array objects are useful when data are mostly contiguous; large intervals of the array will be stored with infrequent gaps in spans of 1000 values or more. Arrays can represent undefined (null) intervals of data. Individual array elements can be erased and replaced with undefined values. Valid array elements are represented by a list of *SimpleInterval* objects (see Figure 2). These track the start and end array indices (inclusive) for non-null, defined data. Array data values are represented by a Java array of the corresponding Java primitive type: `int[]`, `float[]` or `double[]`. Null array elements cannot be represented by the Java null value; a user must check the list of valid intervals to know if an array element is valid. The actual value of the null array element is zero, since this is the value the Java specification requires for uninitialized array elements. *SimpleIntervals* associated with an *Array* instance are kept in order, where `valid[n].end() < valid[n + 1].start()`. If zero valid *SimpleInterval* objects are associated with an *Array*, it means there is no data for the interval specified by *Array.start* and *Array.end* (inclusive) with the specified *ObjectId*.

When reading *Array* objects from the server, there are two modes. In one mode, the server will throw an exception if a specified *ObjectId* does not exist. In the other mode, it will return a placeholder array that will have a value of false for *Array.exists()*. Users can query the available valid intervals for a set of *ObjectIds* without needing to retrieve all the data associated with the *Array*. Data pedigree traceability is implemented by having an ordered list of *TaggedInterval* associated with each *Array* such that `origin[n].end() ≤ origin[n + 1].start()`. The tag of each *TaggedInterval* is used to store the 64-bit pipeline task ID that created the data stored between the *TaggedInterval*'s start and end (inclusive). The data arrays of implementation classes are mutable and not defensively copied; any modifications to them are reflected in the class owning the data array.

Sparse array objects are useful when the data is undefined for most index or key values. There is only one implementation class of sparse arrays at this time: *FloatSparseArray*. This class has 64-bit floating point keys,

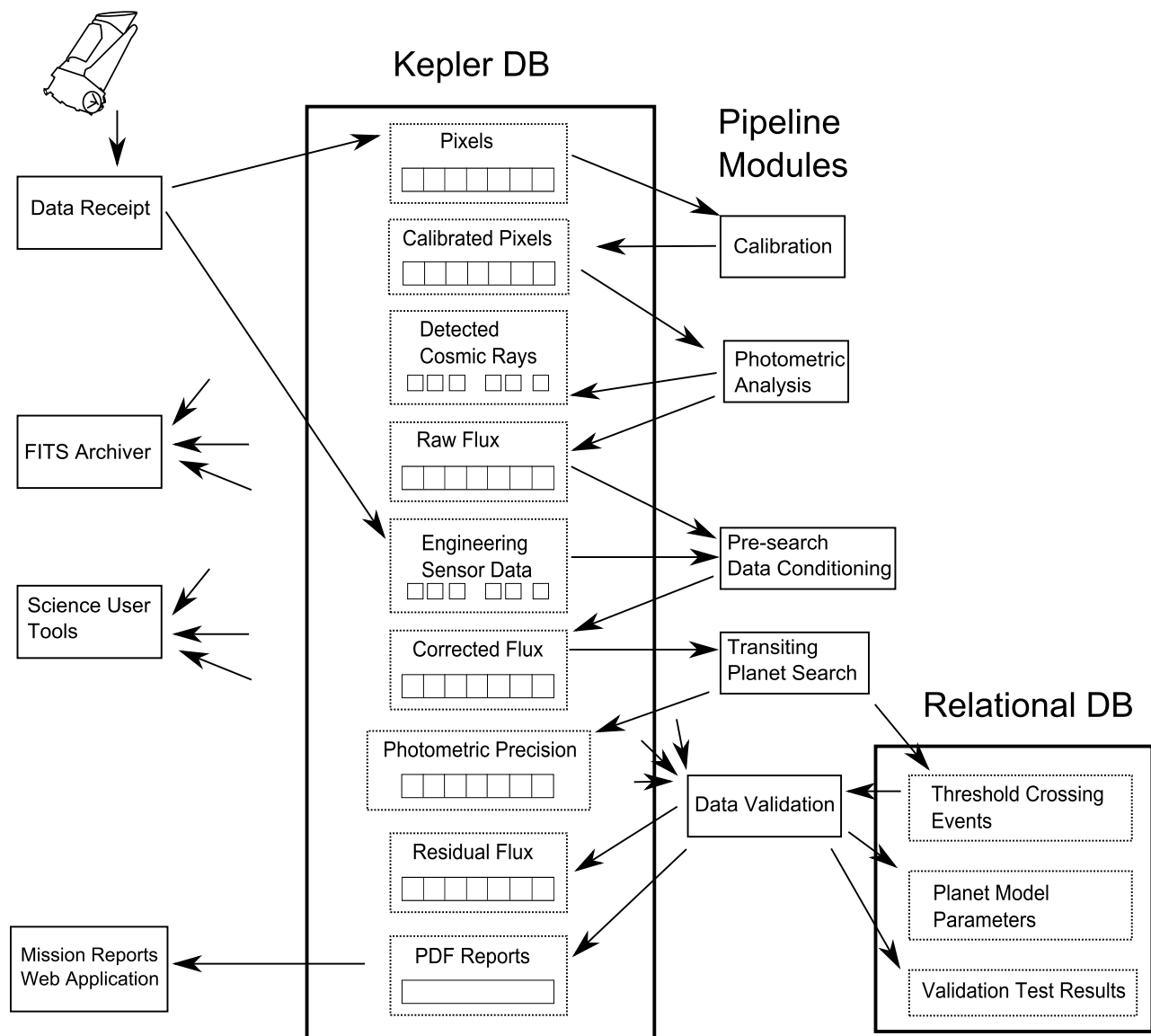


Figure 1. Usage of databases at the *Kepler* SOC. Boxes with solid lines indicate software components (e.g. *Kepler DB*, *Calibration*, *FITS Archiver*). Boxes with dotted lines indicate kinds of data stored (e.g. pixels, raw flux). Within the data boxes, contiguous boxes below the name indicate Arrays, non-contiguous boxes indicate SparseArrays, and solid boxes indicate Blobs. Relational database tables do not have these boxes. Arrows signify the flow of information from source to sink (e.g. pixels are Arrays produced by Data Receipt). Multiple arrows without an obvious source indicate that data comes from many sources within the *Kepler DB*. Many components are omitted between the spacecraft and Data Receipt.

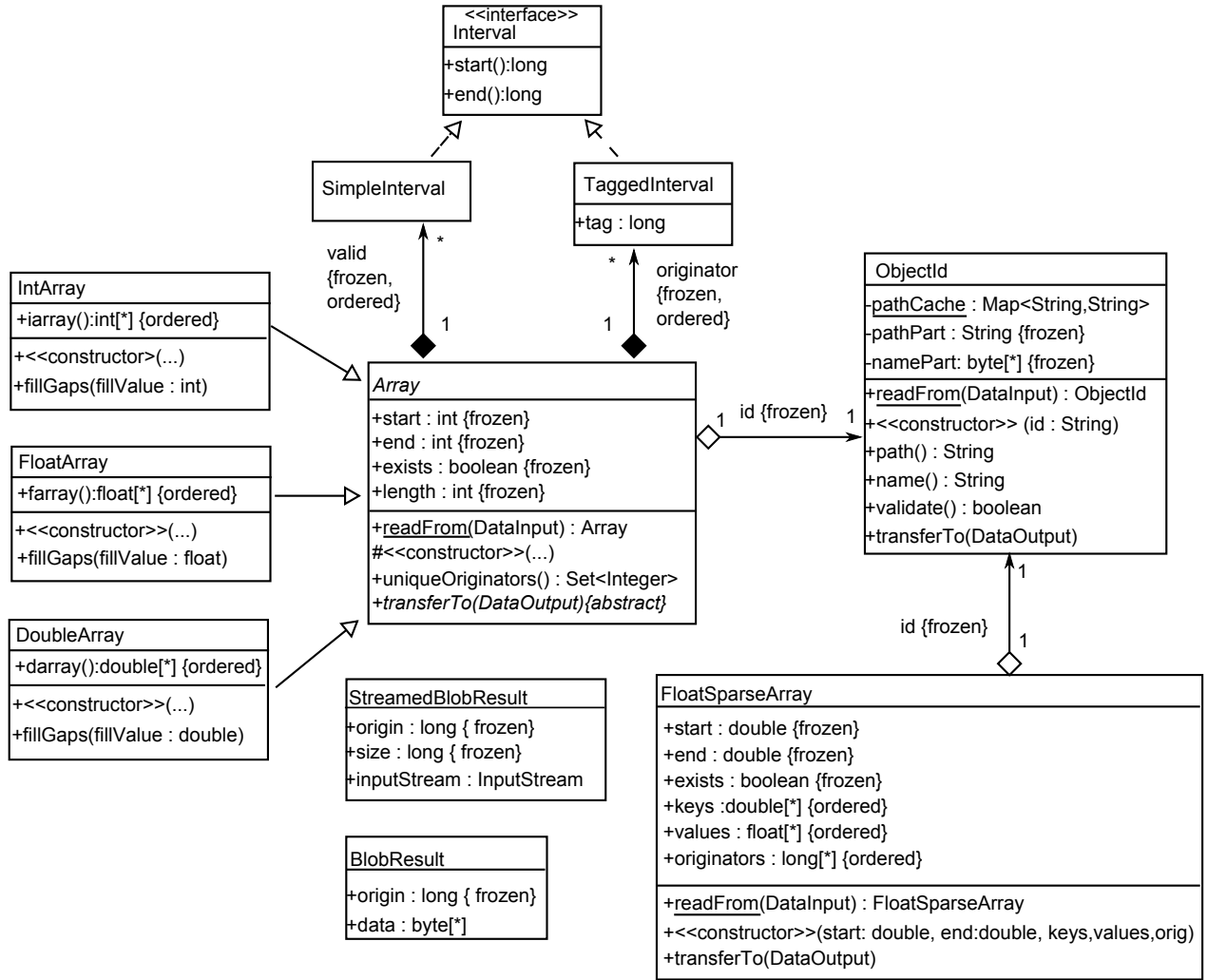


Figure 2. User data types.

32-bit floating point values and 64-bit integers to store pipeline task originators. These are all stored as Java primitive arrays in the FloatSparseArray class. Each Java primitive array is of identical length; the key[i] is the key for value[i]. Likewise, originator[i] is the pipeline task ID which generated value[i]. New data points can be inserted at arbitrary keys in the FloatSparseArray; duplicate keys are not allowed.

The client interface provides basic CRUD functionality for Array, SparseArray, and Blobs. When writing Array and SparseArray objects, there are two modes. One assumes the start and end keys or indices of SparseArray or Array, respectively, are authoritative. In the authoritative mode undefined values are considered as null values; existing data stored for the specified ObjectId between the start and end will no longer be flagged as valid. FloatSparseArrays will have elements between $[FloatSparseArray.start(), FloatSparseArray.end())$ deleted. The other mode is merge mode. In merge mode, nulls in the Array or SparseArray being written are considered unauthoritative. If a valid data value already exists, it will be continue to be flagged as valid for Arrays. For SparseArray new values are inserted or overwrite values with the same key. Merge mode is used to ingest data that arrives out of order from the spacecraft or one of the intermediate organizations that deliver data to the SOC.

Binary large objects (Blobs) are plain files or byte arrays with an attached originator ID. Blobs can be read or written as Java primitive byte arrays, InputStream/OutputStream, or directly with files. Blobs larger than

2Gi-1B must be handled using the streaming methods or transferred using files. A 64-bit integer originator ID is associated with each Blob. When a Blob is written, the originator ID must be supplied. When a Blob is read, the originator ID is returned along with the Blob.

There are methods to query the set of existing ObjectIds currently stored in the database. Queries are similar to UNIX shell file glob expressions, but allow for numerical constraints. So a query “Array@/cal/*” would retrieve all the Array ObjectIds stored under “/cal/*” (i.e. all the calibrated Array ObjectIds). The query “/cal/pixels/sci/lct/ SocCal/10/1/[400,500]:12” would return all the ObjectIds stored for CCD module 10 output 1 in column 12 between rows 400 to 500 inclusive. There is also a method to return all the path parts of the ObjectIds rather than the ObjectIds themselves. This is useful to display the organization of the ObjectIds without enumerating all of them.

A *Kepler DB* command-line interface (CLI) is supplied that can read and write user data objects. Blobs are exported as plain files. Arrays and SparseArray can be exported in a ‘|’ delimited format, an XML format, or MATLAB “.mat” file format. Queries to list existing user data objects can be executed by the CLI.

3.2 Transactionality

The *Kepler DB* maintains the basic transactional database assumptions of atomicity, consistency, isolation, and durability known as ACID,⁴ with some caveats. Modifications of multiple user data objects are atomic; all modifications will succeed or all will fail. Transactions are isolated from changes made by other transactions. The *Kepler DB* implements multi-version concurrency control. This allows transactions to see their own copy of uncommitted, dirty data privately without the need for blocking locks. *Kepler DB* supports the READ COMMITTED isolation level; transactions will see changes made by other transactions once their commit is complete. Consistency ensures the view of the data always satisfies the validity constraints placed upon it. In the context of *Kepler DB*, consistency means that data object metadata must also match the data being stored. The *Kepler DB* supports rolling back (undoing) changes made to user data objects. Should the *Kepler DB* fail in the middle of a transaction it will recover to a consistent state when it is restarted. Durability is the property that once a transaction has been completed data will not be lost. The *Kepler DB* takes steps to guarantee durability across process crashes and similar failures, but not against power failures or other hardware or low-level software errors. For example, it has no way of repairing corrupted disk blocks or recovering from file system errors which might result from power failures.

The *Kepler DB* coordinates transactions with other databases that support XA⁹ distributed transactions; this is implemented by integrating with the standard Java transaction API¹⁰ (JTA). The *Kepler* science processing pipeline uses the JBossTS^{6,12} transaction manager to coordinate distributed transactions with a relational database and a messaging service. Each science processing pipeline worker process has several threads, each of which executes a different, mutually exclusive unit of work. There is one transaction manager per pipeline module worker thread. Local transactions, not involving other data sources, are also supported. Users can begin, commit, and rollback local transactions without involvement of the XA transaction manager. Transactions have user-configurable timeouts, which will trigger an automatic rollback of the transaction that has reached its timeout.

A client thread has thread-local variables that associate it with a transaction identifier (XID) and a connection to the *Kepler DB* server. After this association has been made, child threads inherit a reference to the same XID and connection. If the parent thread ends the transaction and begins a new one, the reference they share still remains pointed at the same XID and connection. In this way, more than one thread can read or write to the *Kepler DB* server in the same transaction. Data Receipt uses this feature to parallelize loading the database where one thread can be parsing FITS files while another is loading the database. This association between parent and child thread can be broken by calling the `disassociateThread()` method on the client. This allows child threads to have their own private transaction context.

3.3 Management Interface

Kepler DB can be managed via the Java Management Extension¹¹ (JMX) GUI, jconsole, as well as other JMX management tools using Java Remote Method Invocation. The JMX management beans exported by the *Kepler DB* server allow for the inspection of currently running transactions, the state of the transactions, the clients’

Internet Protocol address, and when the transaction will timeout. The management user can force the rollback of any transaction. This can be useful if the client disappears and cannot be restarted.

The *Kepler DB* server reports performance metrics to a centralized performance metrics server⁷ as part of the *Kepler Pipeline Framework*.⁷ These performance metrics measure the execution time of various methods. The rate these metrics are reported can be controlled from the JMX console. The level of detail for debug logging statements in the code can be controlled from the JMX console.

A subset of the management functionality which is available via JMX, is also available through the *Kepler DB CLI*.

4. IMPLEMENTATION

4.1 Transactional Files

TransactionalFile is the abstract base class for the on-disk representation of all user data structures in the *Kepler DB* server. TransactionalFile has common locking code and methods used to coordinate transaction state changes. Figure 3 is the class diagram for TransactionalFile and its subclasses. This class does not have any methods to write or read data, or maintain state for individual transactions. Users of TransactionalFile must first call acquireReadLock() and releaseReadLock() if they want to perform more than one operation on a TransactionalFile in a critical section. All subclasses allow multiple threads to read and write concurrently. However, none allow I/O once a transaction is in the process of committing. Subclasses do not cache data; the server relies on the many layers of caching provided by the operating system and storage array. Dirty data is written to a journal which can be used when recovering from a crash.

The read and write lock acquisition methods are implemented using the class ReentrantReadWriteLock in the Java package java.util.concurrent.locks. In addition, TransactionalFile implements a transaction-level lock. Unlike locks from the j.u.c package, or Java monitors, which are owned by threads, the transaction lock is held by a transaction. This means more than one thread can access the same locked TransactionalFile. Only one transaction can hold the transaction lock. See section 4.3 for more information.

TransactionalStreamFile streams data to and from a file. On-disk, it represents its data as a single file with a header that describes the originator of the file. Transactions that have written to a TransactionalStreamFile, but not committed represent their dirty data in a separate file from the committed data file. When a transaction with dirty data commits, the dirty file becomes the committed data file. TransactionalStreamFile is used to implement Blobs. If a transaction attempts to read its own dirty data, then it will get back the contents of its dirty file.

TransactionalRandomAccessFile allows random access reading and writing of bytes. The on-disk representation of TransactionalRandomAccessFile makes use a file for committed data and additional files for uncommitted data (one per transaction that has made modifications). Metadata, the locations of valid data and the originators for data points, is stored in a different file from the actual data. Each ObjectId shares a file with 64 other ObjectIds in what is known as a container file. Uncommitted, dirty data is stored in JournalFile. On disk, each entry in the JournalFile has the ObjectId, the length of the data written and the modified data. TransactionalRandomAccessFile tracks the location of its entries in the JournalFile for when dirty data must be written into the committed data file. If a transaction attempts to read its own dirty data then TransactionalRandomAccessFile merges the journaled data with the committed data.

TransactionalIndexedFile allows a single precision floating point value and its originator ID to be indexed by a double precision number. In the *Kepler* science processing pipeline, this index number is usually the Modified Julian Date of the data point. TransactionalIndexedFile's unit of I/O is a FloatSparseArray object. The on-disk representation uses the container file organization (section 4.2), which allows multiple ObjectIds to share the same file. On top of this representation is a B-tree,⁵ which stores the sparse array as an index. B-trees are efficient, ordered, balanced tree data structures used in many database management systems. Unlike other trees' data structures, (key, value) pairs are stored close to each other on-disk so as to have better locality of reference. As with TransactionalRandomAccessFile, dirty data is written to a JournalFile and merged when a dirty read is required and at commit time.

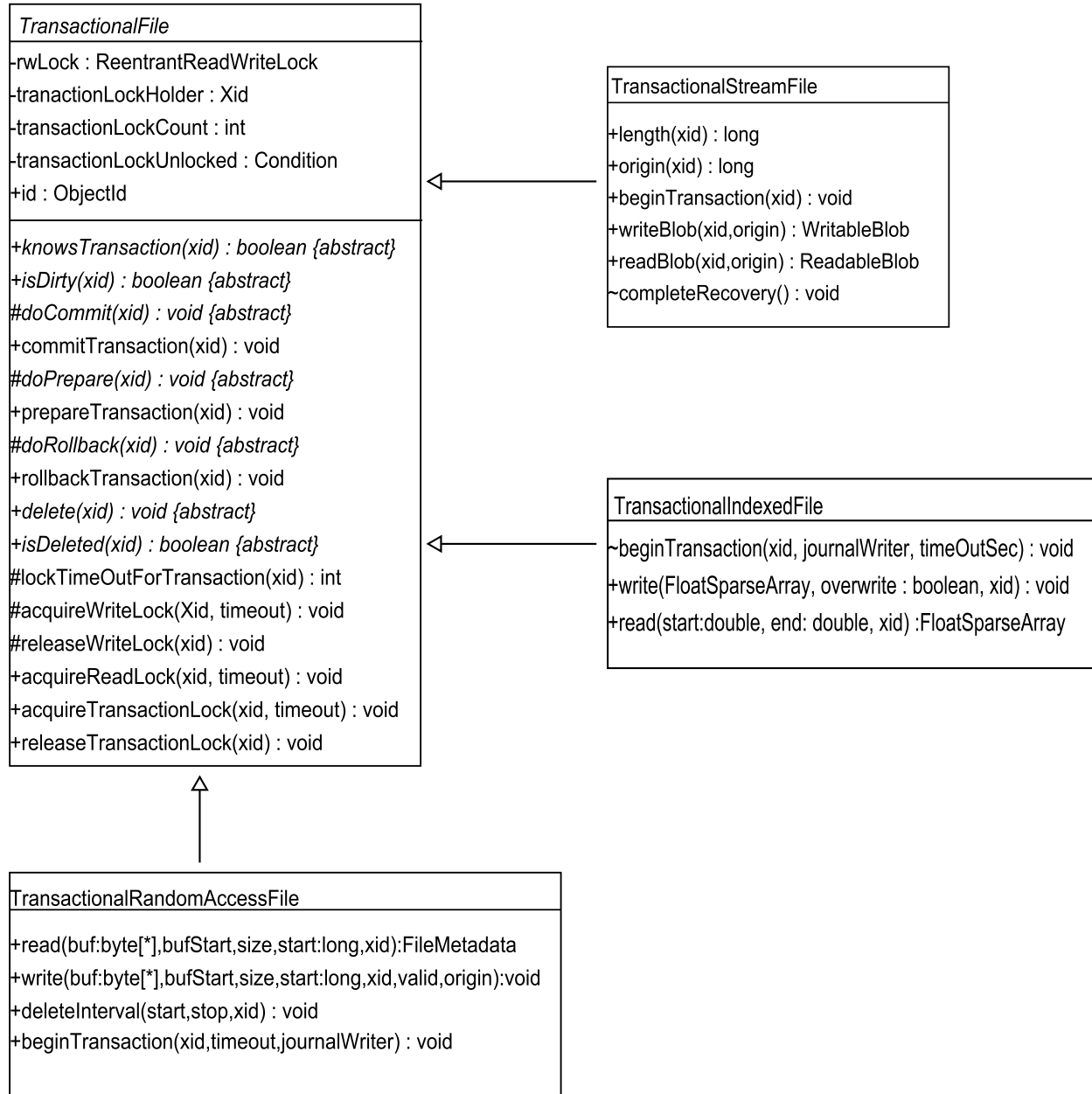


Figure 3. TransactionalFile and subclasses. Xid refers to a transaction identifier.

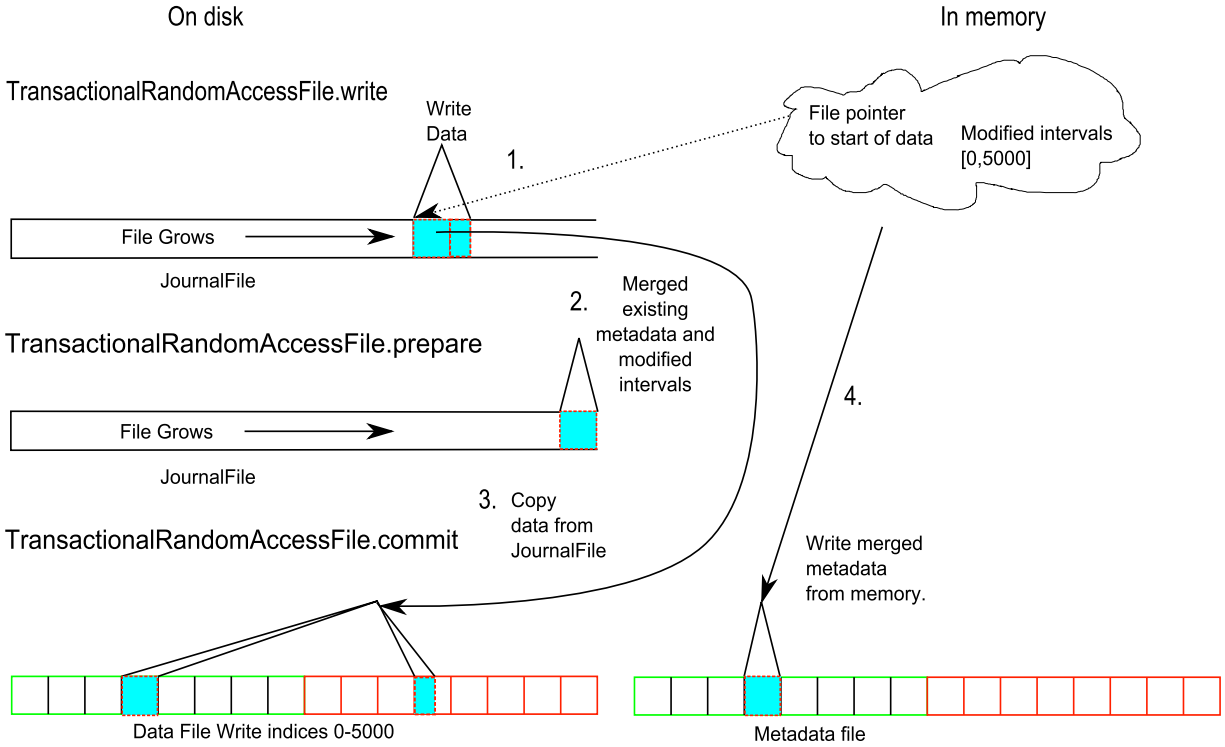


Figure 4. Writing to a container file. Step 1 writes data into the journal file; in memory there is a pointer to the offset in the journal where this change starts. Step 2 creates a combined set of old and new metadata; updated valid indices and originator information. Step 3 commits this data to the container file containing committed data. Step 4 writes the new metadata from memory into a separate file. The data written in Step 2 is only used if the in-memory representation is not available, for example, after a process crash during recovery.

4.2 Storage Allocation

TransactionalRandomAccessFile and TransactionalIndexedFile store their data in a container file. A container file is broken up into 8 KiB blocks where each 8 KiB block is allocated to a specific ObjectId. Up to 64 ObjectIds can share the same file. After the first 8KiB block has been read, the next block for the same ObjectId is located $64 * 8$ KiB away from the first block. In this way, data points stored at the same index or time are stored in roughly the same location on disk. Figure 4 shows the steps of writing data into an Array in a container file.

Large gaps in the continuity of some Arrays exist, caused by target stars now being on other CCD channels after a roll of the spacecraft. The UNIX virtual file system semantics allows for sparse files so that file system blocks that are not written will not be allocated. Unwritten file system blocks intervening with written blocks will not be allocated by the file system, saving some space. TransactionalIndexFile uses the same container file layout for storage allocation, but it writes B-tree nodes instead of raw Array data.

In the file system, container files are numbered starting from zero. Their number is hashed, which results in a directory number where they reside. This prevents directories from containing more files than the file system is capable of handling efficiently. ObjectIds are associated with a particular file and series of blocks in that file using B-tree index files (this is not the same as the TransactionalIndexFile). Each index file has the ObjectId as the key, and the value is the file ID and the associated index in the file containing the first data block assigned to the ObjectId. The path part of the ObjectId is appended to a root directory name of the *Kepler DB* data storage to create a root directory for all the files with the same ObjectId path name. To prevent contention caused by too many threads on too few B-tree index files, naming conventions have been employed in the *Kepler* science processing pipeline to split one kind of data product into one path per process or at most a few processes.

For example, pixels addresses are broken up by CCD module and CCD output. But the pixel address

(*row, column*) is part of the name. Data products with only a few hundred thousands entries per path are not broken up. For example, the raw flux produced by Photometric Analysis simply have a common path for all raw flux; the target number is part of the name.

Data written into a `TransactionalRandomAccessFile` or `TransactionalIndexFile` are journaled in a separate file along with all the modifications for a particular transaction. When a transaction is committed or there is a recovery from a crash, the journal file is read back and data is written into the container file for the `ObjectId` of the `TransactionalRandomAccessFile` or `TransactionalIndexFile`. `TransactionalRandomAccessFile` or `TransactionalIndexFile` record the positions in the `JournalFile`, which contains dirty data so that a transaction can read its own uncommitted data. This implements the multi-version concurrency control for these subclasses of `TransactionalFile`.

4.3 Deadlock Prevention

A transaction can complete in two ways: commit or rollback (undo). XA⁹ transactions use a two-phase approach to commits. In the first phase, the prepare phase, all transactional data sources are required to acquire all the resources needed to commit changes made by a transaction. *Kepler DB* does not allocate the space in the data files. However, all changes made by a transaction are safely stored in journal files or additional files for Blobs. If the database process crashed after the prepare phase, it can still recover and redo the commit. The *Kepler DB* executes a prepare phase even for transactions that are purely local (not started by a distributed transaction manager). During prepare the transaction must acquire the transaction lock on every `TransactionalFile` it has modified. In order to prevent a deadlock, a cycle of locks involving other preparing transactions, an ordering is imposed on the locking of `TransactionalFiles`. Algorithm 1 describes how the transaction lock is implemented and how the ordering is imposed on `TransactionalFiles`.

Algorithm 1 Prepare or rollback locking.

```

 $x := \text{transaction}$ 
 $l := \text{list of transactional files modified by } x$ 
sort  $l$  by their ObjectId
for all  $f := \text{TransactionalFile}$  in  $l$  do
     $w := f$ 's read/write lock
     $u := \text{the condition variable associated with lock } w \text{ that signals the unlock of } f$ 's transaction lock.
     $h := f$ 's transaction lock holder, this is null if the transaction lock is not held
    acquire the write lock  $w$ 
    while  $h \neq \text{null}$  do
        wait on  $u$  {  $w$  has been atomically reacquired after this statement }
    end while
     $h := x$ 
    release  $w$ 
end for
for all  $f := \text{TransactionalFile}$  in  $l$  do {in parallel}
     $w := f$ 's read/write lock
    acquire write lock  $w$ 
     $h := f$ 's transaction lock holder, this is null if the transaction lock is not held
    if  $x \neq h$  then
        release  $w$ 
        throw exception
    end if
    prepare  $f$  or rollback  $f$ 
    release  $w$ 
end for

```

Having a per-transaction lock rather than a per-thread lock allows the prepare and rollback to have a single, deterministic thread for locking, but allows for parallel execution when moving data during the prepare and com-

mit phases. After the prepare phase, data are moved in parallel by calling `commit()` on all the `TransactionalFiles` modified in the transactions. Once this is done, Algorithm 2 is executed to release the transactional lock and allow other transactions access to those `TransactionalFiles`. Readers and writers follow Algorithm 3 in order to interact with other readers and writers as well as threads in the commit or prepare phase. Transactions that do not share `TransactionalFiles` are not dependent on each other and will proceed without blocking on each other's transaction locks.

Algorithm 2 Commit or rollback unlocking.

```

for all  $f := \text{TransactionalFile}$  in  $l$  do {in parallel}
   $w := f$ 's read/write lock
   $u :=$  the condition variable associated with lock  $w$  that signals the unlock of  $f$ 's transaction lock.
   $h := f$ 's transaction lock holder, this is null if the transaction lock is not held
  acquire write lock  $w$ 
   $h := \text{null}$ 
  signal all threads waiting on  $u$ 
  release  $w$ 
end for

```

Algorithm 3 Locking during a read or write to `TransactionalFile`.

```

 $f := \text{TransactionalFile}$  that is the target of an I/O operation.
 $w := f$ 's read/write lock
 $u :=$  the condition variable associated with lock  $w$  that signals the unlock of  $f$ 's transaction lock.
 $h := f$ 's transaction lock holder, this is null if the transaction lock is not held
acquire read lock on  $w$ 
while  $h \neq \text{null}$  do
  wait on  $u$ 
end while
perform read or write operation on  $f$ 
release  $w$ 

```

4.4 Recovery

When the database server starts, it runs the recovery algorithm to put the database back into a consistent state. It may have been in a consistent state before, in which case the recovery algorithm does nothing. If there were live transactions when the database server went down, the recovery algorithm does something different depending on the state of the transaction at the time the server went down. If the transaction had modifications, but had not reached the commit phase, then the journal files (Figure 4) containing those modifications are removed. If the transaction had reached the commit phase then all, some, or none of the prepared data may have been written to permanent files. Data from the journal file are replayed into the committed data files. Each journal entry is associated with an `ObjectId` and a type of `TransactionalFile`. Each journal entry is passed to the transactional file, which applies the change contained in the journal entry. XA transactions require that the transaction state cannot be erased until the transaction manager tells the *Kepler DB* it is safe to remove the transaction state. The outcome of the recovery is recorded until the XA transaction manager asks about the transaction. Records of local transactions are removed after recovery.

5. DEPLOYMENT AND OPERATION

Every cluster in the *Kepler* SOC runs a single instance of the *Kepler DB* server on a Dell R900 with 64GiB of RAM and 2xQuad Core 2.4 Ghz Xeon E7330. The operating system is Fedora 11 GNU/Linux. Database storage is over a 4Gi bit/s fiber channel SAN redundantly connected to a 3PAR S400 storage device. Each 3PAR volume used for storage is configured for RAID 1+0 spread over 188 7.2k RPM SATA disks with a mixture of 1TB and 750GB disks. Redundant SAN connections are managed by the multipathd program, which comes as part of

Fedora 11. When flight operations were beginning, the maximum volume size the S400 was capable of creating was 2TB. These S400 volumes are further aggregated on each host by using Linux Volume Manager (LVM) to present a single unified 8TB volume. This 8TB LVM volume is formatted with the ext3 file system. Before the end of the mission, this file system will reach its capacity. Before that happens, the underlying LVM volume will be enlarged and the file system will be migrated to the ext4 file system.

There are several duplicate clusters used for monthly processing, quarterly processing, flight operations and testing. Typically, flight data is ingested into one of these clusters and a read-only snapshot is made of the base volumes on the S400. To do this, the *Kepler DB* instance running on the cluster is shut down in a consistent state. A snapshot is taken of each S400 volume that composes the LVM volume. A writable snapshot is made of each of the read-only snapshots and then exported into each cluster where it is needed. This same procedure is also used when processed data are desired in each clustered environment.

ACKNOWLEDGMENTS

The authors would like to thank Bill Borucki, David Koch, Jon Jenkins and David Pletcher for their leadership of the *Kepler Mission*. Peter Tenenbaum, Patricia Carroll and Susan Blumenberg provided assistance with preparing this document. Jay P. Gunter and Jason Brittain provided technical assistance. Finally, we would like to thank everyone involved in the Kepler mission; without them none of this would have been possible.

Funding for the *Kepler Mission* is provided by NASA's Science Mission Directorate.

REFERENCES

- [1] Koch, D. G. *et. al.*, "Kepler mission design, realized photometric performance, and early science," *ApJL* 713(2), L79-L86 (2010).
- [2] Middour, Christopher K., "Kepler Science Operations Center Architecture," *Proc. SPIE* 7740, (2010).
- [3] Cudre-Mauroux, P. *et. al.*, "A Demonstration of SciDB: A Science Oriented DBMS," *Proc. VLDB*, (2009).
- [4] Gray, Jim, and Reuter, Andreas, [Transaction Processing: Concepts and Techniques], Morgan Kaufmann, (1993).
- [5] Comer, Douglas, "The Ubiquitous B-Tree," *Computing Surveys* 11(2), (1979).
- [6] Little, Mark *et. al.*, [Java Transaction Processing], Prentice Hall, (2004).
- [7] Klaus, Todd *et. al.*, "The Kepler Science Operations Center pipeline framework," *Proc. SPIE* 7740, (2010).
- [8] Tenenbaum, Peter *et. al.*, "An algorithm for the fitting of planet models to Kepler light curves," *Proc. SPIE* 7740, (2010).
- [9] The Open Group, "Distributed Transaction Processing: The XA Specification," <http://www.opengroup.org/onlinepubs/009680699/toc.pdf>
- [10] Sun Developer Network, "Java Transaction API (JTA)", <http://java.sun.com/javase/technologies/jta/index.jsp>
- [11] Sun Developer Network, "Java Management Extensions (JMX) Technology", <http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement>
- [12] JBoss, "JBoss Transactions", <http://www.jboss.org/jbosstm>